

Java ADT Benchmark (nanoTime).

Warmup ops: 15000, Measure ops: 60000, Trials: 7

Sanity checks: OK

== Stack: ArrayListStack ==

Workload1 bulk push+pop	median:	7.55 ns/op	checksum: 449985000
Workload2 mixed steady-state	median:	21.77 ns/op	checksum: -1055495428

== Stack: DLinkedListStack ==

Workload1 bulk push+pop	median:	17.56 ns/op	checksum: 449985000
Workload2 mixed steady-state	median:	67.86 ns/op	checksum: -1055495428

== Queue: ArrayListQueue ==

Workload1 bulk enq+deq	median:	13.23 ns/op	checksum: 449985000
Workload2 mixed steady-state	median:	27.93 ns/op	checksum: 8738648310

== Queue: DLinkedListQueue ==

Workload1 bulk enq+deq	median:	16.84 ns/op	checksum: 449985000
Workload2 mixed steady-state	median:	30.80 ns/op	checksum: 8738648310

== PriorityQueue: SortedArrayListPQ ==

Workload1 bulk enq+deq (uniform priorities)	median:	3411.72 ns/op	checksum: 449985000
Workload2 mixed steady-state (uniform priorities)	median:	3598.22 ns/op	checksum: -101944034770
Workload3 skewed priorities (bulk)	median:	3969.65 ns/op	checksum: 449985000

== PriorityQueue: SortedDLinkedListPQ ==

Workload1 bulk enq+deq (uniform priorities)	median:	23382.18 ns/op	checksum: 449985000
Workload2 mixed steady-state (uniform priorities)	median:	21523.28 ns/op	checksum: -101944034770
Workload3 skewed priorities (bulk)	median:	25758.79 ns/op	checksum: 449985000

== PriorityQueue: BinaryHeapPQ ==

Workload1 bulk enq+deq (uniform priorities)	median:	73.84 ns/op	checksum: 449985000
Workload2 mixed steady-state (uniform priorities)	median:	59.41 ns/op	checksum: -106337492798
Workload3 skewed priorities (bulk)	median:	55.58 ns/op	checksum: 449985000

The benchmark compares ArrayList-based and doubly linked list based implementations of Stack, Queue, and PriorityQueue using `System.nanoTime()`. Each workload had 15,000 warmup operations, 60,000 measured operations, and 7 trials. All sanity checks passed, so the implementations are correct. The results show that Big-O complexity matters, but real machine effects like caching and memory layout also have a big impact.

For the Stack implementations, both `ArrayListStack` and `DLinkedListStack` have $O(1)$ push and pop operations. Based on Big-O alone, they should perform about the same. However, the ArrayList version is much faster (7.55 ns/op in bulk) compared to the linked list version (17.56 ns/op). The main reason is memory layout. ArrayList stores elements in contiguous memory, which makes it very cache-friendly. The CPU can access elements quickly because they are next to each other. A linked list, on the other hand, stores each element in a separate node object somewhere in memory. Each operation involves following pointers, which can cause cache misses and slow things down. So even though both are $O(1)$, the array-based version wins because it works better with modern hardware.

The Queue implementations show a similar pattern. Both the circular-buffer `ArrayListQueue` and `DLinkedListQueue` have $O(1)$ enqueue and dequeue operations. Since the array queue avoids shifting by using a circular buffer, it keeps constant-time performance. The `ArrayListQueue` is still slightly faster (13.23 ns/op vs. 16.84 ns/op bulk), again because of better cache locality and fewer object allocations. The difference isn't as big as with stacks, but the array version is still consistently faster.

The biggest differences show up in the PriorityQueue implementations. The sorted ArrayList and sorted linked list both keep their elements in order at all times. That means every

time a new item is added, the program has to find the correct spot for it. If the list is already large, that can take a while.

For the sorted ArrayList version, when a new element is inserted near the front, all the other elements have to shift over one position to make space. If the list has thousands of elements, that's thousands of moves for just one insert. The sorted linked list doesn't shift elements, but it has to walk through the list one node at a time to find the correct position. That "walking through" process is slow because it follows pointers in memory. In the results, the linked list version is much slower than the array version because pointer chasing is more expensive than shifting elements in a continuous block of memory.

The BinaryHeapPriorityQueue performs much better. Instead of keeping everything perfectly sorted, it keeps the data in a special tree-like structure stored inside an array. When you insert or remove an element, it only needs to swap a small number of elements to keep the structure valid. Even when the structure gets large, the number of swaps stays relatively small. That's why its times (around 55–74 ns/op) are dramatically lower than the sorted versions. The heap also benefits from using an array, so it gets the same cache advantages as the other array-based structures. Overall, the sorted versions slow down a lot as more elements are added because each insertion becomes more expensive. The heap scales much better because it avoids large shifts or long pointer traversals. That's why it clearly dominates for larger workloads.